



1.9. Объекты и классы

Привет!

На связи шпаргалка урока 1.9. Объекты и классы, в которой вы найдете информацию по теме урока и полезные советы, которые помогут выполнить домашнее задание.

Время прочтения: 18 минут

Наконец настал тот день, когда вы перестанете вздрагивать при виде слов «класс» и «объект».

Java является объектно-ориентированным языком. Это означает, что его философия подразумевает любые сущности объектами, которые взаимодействуют друг с другом.

Хотите проверить, что текст написан на русском? Создаете объект Переводчик. Нужно напечатать текст? Пишем объект Принтер. Захотелось посчитать число пи? Объект Калькулятор так и напрашивается. Хотите сначала прочитать что-то из файла, а затем отправить на email то, что было прочитано? Создаете объект Читатель и объект Почтальон. Пишите, что Читатель должен прочитать файл, а потом текст передать Почтальону, который уже сам должен разбираться, как и куда ему отправить этот текст. Каждый объект имеет свою зону ответственности и должен отвечать исключительно за одну какую-то обязанность.

Принтер должен печатать, почтальон отправлять на почту данные, читатель читать, а калькулятор считать. Принтер не должен уметь читать файлы, а почтальон переводить тексты. В этом вся Java.

Так что же такое объект, если простыми словами?

Всё, что выглядит немного сложнее и/или умеет немного больше, чем просто цифра или символ, будет являться объектом. Объекты имеют как состояние (характеристики или свойства), так и поведение (умения, навыки или просто методы).

Вы уже знакомы с такими объектами, как массив и строка.

И массив, и строка имеют в себе какие-то характеристики (в случае с массивом это переменная `length`) и/или умения, т. е. методы (в случае со строкой `toUpperCase`, `replace` и т. д.).

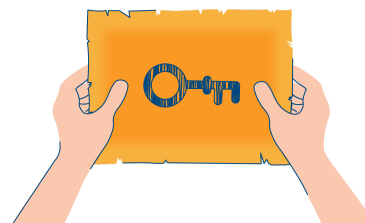
А где здесь класс? Начинаем разбираться.

Класс и объект

Чтобы объяснить, что это такое класс и объект и как они связаны, вспомним ситуацию из всем известного фильма про самого обаятельного пирата Голливуда — Джонни Деппа.

В одной из частей «Пиратов Карибского моря» наш герой достает рисунок ключа и заявляет, что самого ключа у него нет, но есть «лучше», а именно рисунок ключа.

Если отталкиваться от этой ситуации, то класс — это рисунок ключа, а объект — сам ключ.



Чтобы создать ключ, нам нужен рисунок. На рисунке указаны общие данные для ключа, его размер и форма. Если мы собираемся создать этот ключ, мы должны следовать этим характеристикам.

Еще один пример. Чтобы построить дом, нам нужно иметь чертеж дома, передать его строителям, а они уже по чертежу начнут ставить стены, укладывать пол и наращивать этажи. Согласитесь, что словесные указания сотрудникам «построй дом» вряд ли позволят нам получить ожидаемый результат.

В обоих примерах мы имеем две сущности, а именно чертеж и результат.

Класс является этим самым чертежом. По сути, он представляет собой план, по которому собирается объект. Также класс содержит все общие свойства, которые не могут отличаться у любых объектов по данному чертежу. В случае с ключом ключ не может быть другого размера или формы, следовательно, размер и форма, которые подразумеваются чертежом, будут общими характеристиками для всех.

Объект же является тем самым результатом. И он тоже может хранить свои характеристики, которые будут уникальными только для него или для нескольких объектов из общей массы. В случае с ключом, например, уникальная характеристика непосредственно объекта может быть форма рукоятки или, допустим, сорт металла. Чертеж (класс) не обязывает нас делать площадку для руки открывающего

определенной формы, так как это не повлияет на возможность отпирания двери. Или не заставит нас отливать ключ из конкретного металла, так как это тоже вряд ли может повлиять на способность отпереть дверь.

Умениями (методами) такого объекта стали бы, как минимум, действия «открыть дверь» и «закрыть дверь», а сам объект двери бы передавался этим методам в виде параметра.

Класс в Java

Что же представляет собой класс в Java?

Это сущность, которая описывает состояние (переменные), поведение (методы) и способы создания своих объектов (если они подразумеваются).

Вы уже писали свой самый простой класс, когда делали домашние задания.

Это был класс на один статический метод (main), без поведения (переменных класса или полей) и необходимости строить объекты этого класса в дальнейшем.

Чтобы объявить класс, нам нужно написать ключевое слово `class` и поставить перед ним модификатор доступа. Обычно это `public`. После слова `class` мы пишем его название и открываем скобки `{}`.

На этом класс создан, и мы можем начать с ним работать.

Классы бывают разных типов, и вот несколько из них:

1. Хранитель данных (обычно эти классы не имеют «умений» и созданы только для агрегации в себе некоторых данных). Например, объект Книга. У нее могут быть свойства или состояние (название, год выпуска, автор, номер издания и т. д.), но все ее «умения» заключаются только в том, что вы можете ее открыть и взять текст, информацию об авторе, годе издания и т. д.
2. Утилити-класс (обычно эти классы имеют только «умения» и созданы для работы с другими объектами, например класс `Arrays` и массивы).
3. Самодостаточные классы (как `String`), которые имеют и состояние (массив символов, в виде которого хранятся данные строки), и поведение (методы, которые с этим состоянием работают, как `replace`, например).

Попробуем реализовать наш собственный утилити-класс по аналогии с `Arrays`.

Он будет предоставлять нам функционал по работе с телефонными номерами, т. е. содержать внутри себя статические методы, которые мы можем в любой части нашей программы вызвать и привести номер телефона в нужный нам вид.

Реализовывать мы будем следующие методы:

1. Удалить лишние символы из номера (дефисы и скобки, если они имеются).
2. Добавить +7 к номеру, если первая стоит 8.
3. Определение оператора (нужно получить три цифры после 8 или +7 и на их основе определить оператора).

Сразу напомним, что имена классов в Java пишутся в UpperCamelCase, т. е. начинаются с большой буквы и каждое следующее слово в имени тоже начинается с большой буквы.

▼ Код тут

```
public class PhoneNumberUtility {
    public static String removeUselessSymbols(String phoneNumber) {
        return phoneNumber.replace("-", "")
            .replace("(", "")
            .replace(")", "");
    }

    public static String validateCountryCode(String phoneNumber) {
        if (phoneNumber.startsWith("8")) {
            return phoneNumber.replaceFirst("8", "+7");
        } else if (phoneNumber.startsWith("7")) {
            return "+" + phoneNumber;
        }
        return phoneNumber;
    }

    public static void printOperatorByPhone(String phoneNumber) {
        String operatorCode = null;
        if (phoneNumber.startsWith("8")) {
            operatorCode = phoneNumber.substring(1, 4);
        } else if (phoneNumber.startsWith("+7")) {
            operatorCode = phoneNumber.substring(2, 5);
        }

        if (operatorCode == null) {
            System.out.println("Invalid phone number");
            return;
        }

        switch (operatorCode) {
            case "916":
                System.out.println("MTS");
                break;
            case "925":
                System.out.println("MegaFon");
                break;
            case "903":
                System.out.println("Beeline");
            }
    }
}
```

```
}  
}
```

Теперь мы можем в любой части программы, будь это метод `main` в другом классе, пакете (папке) или любой другой метод в рамках нашей программы, вызвать наш утилита-класс, передать ему в качестве параметра номер телефона и, например, удалить из него дефисы и круглые скобки.

▼ Это будет выглядеть следующим образом.

```
public static void main(String[] args) {  
    String phoneNumber = "8(916)-111-11-11";  
    System.out.println(PhoneNumberUtility.removeUselessSymbols(phoneNumber));  
    System.out.println(PhoneNumberUtility.validateCountryCode(phoneNumber));  
}
```

Вывод:

89161111111

+7(916)-111-11-11

Как мы помним, из классов создаются объекты.

Но объект утилита-класса нам создавать не требуется, так как в данном случае класс выступает исключительно для того, чтобы позволить нам объявить методы (так как все методы должны быть объявлены внутри класса) и группировать их под общим названием.

Но как нам запретить создавать объекты нашего класса (или, другими словами, экземпляры класса)?

Чтобы создать объект, нам нужно написать в данной ситуации:

```
PhoneNumberUtility object; // Здесь мы объявляем тип переменной и ее имя  
object = new PhoneNumberUtility(); // Инициализируем переменную объектом нашего класса  
  
PhoneNumberUtility utilityObject = new PhoneNumberUtility();
```

Как видите, в конце строки стоят круглые скобки. В Java наличие круглых скобок говорит прямо, что перед нами метод.

Но ведь мы знаем, что методы пишутся с маленькой буквы.

Перед нами особенный метод, который создан для того, чтобы инициализировать наш объект.

Он называется **конструктор**. Его задача в том, чтобы в случае необходимости создать объект, мы обязаны корректно инициализировать его.

Допустим, нам нужно создать объект Читатель, который был упомянут выше. Но он не должен быть создан в том случае, если ему читать нечего. Поэтому при создании объекта Читатель мы должны создать конструирующий метод, который будет жестко требовать от пользователя передать источник (в данном случае файл) данных.

Файла нет — нет и Читателя.

Как же нам ограничить возможность создания объекта нашего утилита-класса? Сделать метод-конструктор приватным. В уроке по методам мы мимоходом познакомились с модификаторами доступа. Сейчас мы снова с ними сталкиваемся.

Конструктор создается следующим образом. Мы пишем стандартную сигнатуру метода, но без возвращаемого значения, а имя метода должно соответствовать имени класса.


Т. е. нужно добавить в наш класс следующее:

```
private PhoneNumberUtility() {  
} // Блок внутри скобок пустой, так как никакой логики при инициализации объекта нам не требуется
```

И это полностью решает возникшую проблему.

Теперь, если мы снова попытаемся создать объект нашего утилита-класса, Java выдаст ошибку.

```
PhoneNumberUtility utilityObject = new PhoneNumberUtility();
```

A screenshot of a code editor showing the line `new PhoneNumberUtility();`. The text is highlighted in a dark background. A red squiggly line is drawn under the entire expression, indicating a compilation error. The word 'new' is in orange, and 'PhoneNumberUtility()' is in blue.

Создание объекта будет подчеркнуто красным, а IDEA выдаст ошибку, что конструктор имеет private access.

Конструирующие методы / Конструкторы

Как мы уже выяснили ранее, конструкторы нужны для того, чтобы объекты не могли быть созданы некорректно.

Обычно они не содержат какую-то логику и просто инициализируют состояние, т. е. переменные класса. Но если переменных у класса нет, то они управляют возможностью создавать объект, как было в упомянутом ранее случае.

Конструкторы подчиняются тем же правилам, что и обычные методы. Разница в том, что конструкторы не могут быть вызваны в произвольное время, а вызываются именно при создании объекта.

Разница еще и в том, что у конструкторов нет возвращаемого значения, так как они, по сути, всегда возвращают экземпляр класса (объект) того типа, который их объявил.

Статическими конструкторы тоже быть не могут, ведь статические методы принадлежат классу, а не объекту.

Попробуем разобраться с конструкторами на примере второго типа класса. Это так называемый дата-класс, он же хранитель данных из списка выше.

У него есть состояние (переменные), но нет никакого поведения (методы).

Создадим класс Книга, который хранит в себе данные об авторе, количестве страниц и издательстве.

▼ Код тут

```
public class Book {
    String authorName;
    int pageAmount;
    String publisherName;

    public Book(String authorName, int pageAmount, String publisherName) {
        this.authorName = authorName;
        this.pageAmount = pageAmount;
        this.publisherName = publisherName;
    }
}
```

Чтобы создать объект Книгу, нам требуется вызвать этот самый метод-конструктор после ключевого слова `new`, а в параметры передать данные, что указаны в сигнатуре конструктора.

```
Book book = new Book("S. King", 100, "Book Publishing Ltd.");
```

В этом случае мы создаем переменную по имени `book`, а его переменным присваиваются значения, которые переданы в конструктор в качестве параметров.

Теперь мы можем обратиться к данным этой книги известным нам способом.

book.authorName вернет нам строку "S. King".

book.pageAmonth вернет нам число 100.

book.publisherName вернет нам строку "Book Publishing Ltd".

Этим переменным мы прямо здесь можем присвоить значения, а не только получить текущие.

```
book.authorName = "L. Tolstoy"
```

Данная строка сменит автора на Льва Толстого.

Но что будет, если мы не создадим внутри класса конструктор? Сможем ли мы создать объект книги?

Сможем, так как в тех случаях, когда мы не создаем конструктор сами, его создает Java. Именно поэтому в самом первом случае с нашим утилити-классом мы смогли создать конструктор.

Стандартный конструктор, что генерируется языком в случае отсутствия объявленных нами конструкторов, выглядит достаточно примитивно и никакие переменные класса не инициализирует.

Выглядит он так:

```
public Book() {  
}
```

Этот конструктор просто позволяет нам создать объект, но при этом все его переменные будут равны стандартным значениям.

Их мы уже упоминали в шпаргалке по массивам.

В том случае, когда вы объявили свой конструктор, стандартный конструктор Java не добавит, т. е. класс можно будет создать только по тем конструкторам, что объявили вы.

Но что делать, если у нашей книги нет издателя?

Есть два решения:

1. Мы можем обязать пользователя постоянно передавать в параметр *publisherName* значение *null*, что является не очень надежным решением.
2. Мы можем написать второй конструктор, который заполняет *publisherName* по умолчанию. В этой ситуации пользователь может передать два параметра

в конструктор, а не три, а конструктор с двумя параметрами уже или инициализирует значения сам, или делегирует эту обязанность конструктору с тремя параметрами. Посмотрим это решение на примере.

Нужно добавить еще один конструктор в наш класс Book.

```
public Book(String authorName, int pageAmount) {  
    this(authorName, pageAmount, null);  
}
```

Про ключевое слово `this` мы поговорим в следующем разделе, но если вкратце, мы просто делегировали выполнение функций конструктору с тремя параметрами, передав в него два пришедших параметра, и один задали сами.

Этим же способом вместо `null` мы можем подать какое-то дефолтное значение для переменной `publisherName`. Допустим, "No publisher".

```
public Book(String authorName, int pageAmount) {  
    this(authorName, pageAmount, "No publisher");  
}
```

Но сразу стоит вспомнить методы и учесть, что уникальность методов (а конструктор является методом) соблюдается за счет сигнатуры (возвращаемый тип, имя, параметры). Так как возвращаемый тип конструктора всегда один и имя тоже, нам остается поддерживать уникальность на уровне параметров.

А значит, метод с двумя параметрами `String` и `int` допускается только один, и мы должны выбрать, передавать третьим параметром `null` или какое-то дефолтное значение.

Ключевое слово `this`

Как вы могли заметить ранее, мы повсеместно в конструкторах применяем ключевое слово `this`.

`this` является переменной, которая всегда ссылается на ваш конкретный объект.

В случае вызова в конструкторе из примера выше следующей структуры:

```
this(authorName, pageAmount, "No publisher");
```

мы, по сути, говорим, что нужно вызвать другой конструктор этого же класса, но с тремя параметрами.

В случае применения конструкции вида `this.authorName = authorName` из примера выше ключевое слово `this` используется для того, чтобы явно показать, к какой конкретно переменной мы обращаемся.

`this.authorName` говорит нам, что нужно взять именно переменную объекта `authorName` и присвоить ей значение параметра `authorName`.

В случае, когда параметр и переменная класса имеют разные имена, слово `this` можно опустить.

```
public Book(String _authorName, int _pageAmount, String _publisherName) {  
    authorName = _authorName;  
    pageAmount = _pageAmount;  
    publisherName = _publisherName;  
}
```

Однако рекомендуется использовать способ с `this`, так как имена параметров IDEA проставляет нам в виде подсказок, и если они написаны с чертой, выглядит это не очень красиво.

```
10 main(String[] args) {  
    new Book(_authorName: "S. King", _pageAmount: 100, _publisherName: "Book Publishing Ltd.");  
}
```

Переменные класса (поля)

Как мы с вами упоминали выше, классы могут иметь состояние.

Это переменные класса или, как их принято называть, поля класса.

Их область видимости ограничивается классом, где они объявлены (т. е. работать с ними можно или при обращении извне в виде `имя_объекта.имя_поля`, или внутри класса (в его методах) с помощью обращения по имени без дополнительных конструкций).

Однако следует помнить, что если поле класса совпадает по имени с параметром метода, необходимо использовать `this.имя_поля` для обращения к полю, так как параметр или локальная переменная (объявлена в конкретном методе с тем же именем, что и поле) будут «затенять» поле.

Также поля могут иметь модификаторы доступа, которые работают по тому же принципу, что и у методов, т. е. поле `public` будет доступно из любого другого метода при обращении `имя_объекта.имя_поля`, а вот к `private`-полю уже нельзя будет обратиться с помощью той же конструкции вида `имя_объекта.имя_поля`.

Вспомним пример с ключом и рисунком ключа.

У класса могут быть поля, которые будут общими для всех его объектов.

Например, размер ключа и его «рисунок» язычков замка не должен быть разным для всех ключей, которые созданы по этому рисунку.

Общие для всех объектов класса данные объявляются статическими полями, т. е. в случае с размером мы должны создать поле следующего вида:

```
public static final int KEY_SIZE = 5;  
// Условный размер, не стоит обращать внимание на значение
```

Важно заметить, что `static final` поля являются константами (значения переменных с модификатором `final` нельзя изменять) и пишутся капсом с разделением в виде `'_'` между словами.

Если нам нужно, например, написать счетчик ключей, который будет общий для всех ключей по нашему рисунку, мы объявляем статическое поле, которое не является константой, так как должно изменяться в случае создания нового ключа.

```
public static int keyCounter = 0;
```

А затем это значение в конструкторе необходимо инкрементировать.

Получится, что счетчик будет увеличиваться каждый раз, когда создается ключ (вызывается конструктор), и его значение всегда будет доступно всем его объектам в корректном виде.

Чтобы обратиться к статическому полю, если оно `public`, нужно использовать вместо `имя_объекта.имя_поля` конструкцию `имя_класса.имя_поля`.

В случае нестатических полей всё работает так же, как и с локальными переменными (которые объявляются в методе), только эти переменные доступны не в одном вашем методе, а во всех методах класса.

Нужно запомнить, что нестатические поля могут различаться для каждого из объектов и именно их значения мы инициализируем в конструкторе.

Статические же переменные мы обычно задаем вручную (инициализируем) и меняем (в случае со счетчиком) внутри класса.

Также следует помнить ту разницу, что в случае локальных переменных при объявлении, допустим, `int i` и обращении к переменной `i` Java (в лице IDEA) попросит

инициализировать переменную. Т. е. перед работой с локальной переменной *i* нужно ей задать значение.

В случае с полями, которые не были инициализированы явно, им присваивается значение по умолчанию (как было в элементах массива при его создании).

Напомним эти значения.

Таблица значений переменных по умолчанию

<u>Aa</u> Тип переменной	<u>≡</u> Значение по умолчанию
<u>short</u>	0
<u>boolean</u>	false
<u>String</u> (или любой другой объект)	null
<u>double</u>	0.0d
<u>long</u>	0L
<u>int</u>	0
<u>float</u>	0.0f
<u>char</u>	0
<u>byte</u>	0

Инкапсуляция

Инкапсуляция является одним из принципов ООП и, если вкратце, говорит нам, что не нужно давать пользователям наших самописных типов (классов и объектов) больше возможностей, чем нужно для выполнения конкретных действий.

Именно для этого существуют модификаторы доступа.

Пользователь не должен иметь доступ к методам, которые ему не положено вызывать, или к полям, которые ему запрещено читать или даже изменять.

Например, возьмем наш объект Книга. Дав возможность напрямую обращаться к полю Название, пользователь сможет после создания изменить название, что будет совершенно некорректно, ведь будет плохо, если наша «Война и мир» под авторством Льва Толстого внезапно станет «Винни-Пухом» под авторством того же Льва Толстого.



Если эта книга используется в нескольких местах программы (предположим, в списке доступных в библиотеке книг и в реестре «долгов» по книгам, взятым в библиотеке), окажется, что человек нам задолжал не «Войну и мир», а какого-то внезапно появившегося «Винни-Пуха».

Именно поэтому мы должны закрыть пользователю доступ для обращения к полю напрямую, объявив поле с модификатором доступа `private`.

Но как тогда дать пользователю возможность прочитать из книги ее имя? Для этого существуют такие методы, как геттеры и сеттеры. Это, по сути, обертка над полем, которая позволяет получить его значение (в случае геттера) или присвоить ему новое значение (в случае сеттера).

О них поговорим в следующем разделе.

Геттеры и сеттеры

Геттеры и сеттеры, как мы уже упоминали, регулируют доступ к приватному полю. В случае необходимости дать доступ на чтение, но закрыть доступ на изменение создается только геттер, а сеттер — нет.

В случае, когда нужно предоставить полный доступ к полю (и чтение, и изменение), создаются оба метода.

Имена геттеров и сеттеров обычно строятся следующим образом: глагол `get/set` и имя поля. Здесь стоит вспомнить, что имена методов начинаются с маленькой буквы, первое слово должно являться глаголом, а каждое новое слово в этом имени начинается с большой буквы.

Объявим геттеры и сеттеры для полей нашей книги.

Автор и название не должны изменяться после создания (т. е. после инициализации полей в конструкторе), а вот издательство поменять можно, ведь книга может начать издаваться в другом месте. Поэтому для первых двух мы создадим только геттеры, а для третьего — еще и сеттер.

▼ Код тут

```
public String getAuthorName() {
    return authorName;
}

public int getPageAmount() {
    return pageAmount;
}

public String getPublisherName() {
```

```

    return publisherName;
}

public void setPublisherName(String publisherName) {
    this.publisherName = publisherName;
}

```

Заметьте, что в сеттерах тоже используется `this` для обращения к нашему полю, так как параметр принято называть тем же именем.

Для работы со статическими полями нам нужно объявлять статические геттеры и сеттеры, т. е. добавить ключевое слово `static` после модификатора доступа.

Для констант (`static final` поля) создание геттеров и сеттеров не применяется, так как их изменяемость заблокирована модификатором `final`.

Доступ к ним (для чтения) регулируется модификатором `public/private`. И обращение идет по схеме `имя_класса.ИМЯ_КОНСТАНТЫ`.

Но что, если сеттить нужно не бездумно, а, допустим, требуется добавить валидацию передаваемого в сеттер значения для установки в поле?

Допустим, нашей книге потребовалась доработка, которая вводит еще данные о годе издания книги. Книга может быть издана в разных партиях, которые могут выходить в разные годы. Потому нам, возможно, в какой-то из моментов понадобится «обновить» данные нашей книги о годе издания.

Для этого мы добавляем поле `publishingYear` типа `int`, добавляем его в наши конструкторы и создаем геттер и сеттер.

Но в сеттере мы должны добавить проверку, что передан корректный год публикации.

▼ В этой ситуации наш сеттер должен выглядеть следующим образом.

```

public void setPublishingYear(int publishingYear) {
    if (publishingYear < 1950 || publishingYear > 2050) {
        System.out.println("Invalid publishing year parameter: " + publishingYear);
        return;
    }
    this.publishingYear = publishingYear;
}

```

Мы проверяем, что переданный год является в допустимых пределах, и устанавливаем переменную.

▼ Результат работы по созданию класса Книга.

```

public class Book {
    private final String authorName;
    private final int pageAmount;
    private String publisherName;
    private int publishingYear;

    public Book(String authorName, int pageAmount, String publisherName, int publishingYear) {
        this.authorName = authorName;
        this.pageAmount = pageAmount;
        this.publisherName = publisherName;
        this.publishingYear = publishingYear;
    }

    public String getAuthorName() {
        return authorName;
    }

    public int getPageAmount() {
        return pageAmount;
    }

    public String getPublisherName() {
        return publisherName;
    }

    public void setPublisherName(String publisherName) {
        this.publisherName = publisherName;
    }

    public int getPublishingYear() {
        return publishingYear;
    }

    public void setPublishingYear(int publishingYear) {
        if (publishingYear < 1950 || publishingYear > 2050) {
            System.out.println("Invalid publishing year parameter: " + publishingYear);
            return;
        }
        this.publishingYear = publishingYear;
    }
}

```

Помимо геттеров и сеттеров, класс может иметь нестатические методы по работе со своими полями или, как в случае с `PhoneNumberUtility`, по работе с параметрами, что приходят в методы.

Пример: класс *String*.

Его код вы можете открыть путем зажатия клавиши `Ctrl` и клика на слово `String` при объявлении переменной этого типа. Или поставить курсор на это слово и нажать `Ctrl + B`.

Шпаргалка урока в PDF-формате:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/361fb197-3d7c-4f52-a1ab-c2d68dc6f69b/1.9.____.pdf